A-05

メモリアクセス高速化のための回路自動生成の一手法
A Circuit Synthesis for High Speed Memory Access in System LSI

岸田和也‡　　神戸尚志†
Kazuya Kishida　　Takashi Kambe

1．はじめに

システム LSI は各種電子機器に広く利用され、年々複雑化・高度化が進み、設計の危機と言われている。特に、急激に複雑・高度化する各種アプリケーションは大規模なメモリやデータを高速にアクセスすることが多く、メモリアクセス部のアーキテクチャ設計が重要課題となっている。

本文では、C 言語ベース設計(Bach システム)におけるメモリアクセス部のアーキテクチャ自動最適化手法(単変数に対するレジスタ化、ループ構造によるメモリアクセスに対するオンチップ配列化)を提案し、実際に回路に反映し、その有効性を確認する。

2．メモリアクセス部のアーキテクチャ最適化手法

システム LSI におけるメモリアクセスの重要な点は、メモリアクセス時間がシステム LSI 全体の処理に対し、ボトルネックとならないことである。対処法として、高速にアクセスしたいデータをオンチップに保持する方法がよく用いられる。しかし、チップサイズを増大させないためにはそのメモリサイズを最小化する必要がある。そこで、本文では 2 つの手法を提案する。第 1 に頻繁に再利用される単変数単位のレジスタ化によるメモリアクセスの高速化を図る。第 2 に再利用される複数データのオンチップ配列化によるメモリアクセスの高速化を考える。

2.1.　再利用データのレジスタ化

オフチップメモリへの頻繁なアクセスがある場合は毎回オフチップメモリをアクセスするのではなく(図 1(a))、レジスタを用意し、オフチップメモリのデータを一時的にレジスタに格納する。これによりデータの読み込み先をレジスタへ変え、オフチップメモリへのアクセス回数を削減する方法を提案する(図 1 (b))。以降、この手法を「レジスタ化」と呼ぶ。

メモリに保存されている値が変更されていない間はレジスタ化が有効であるが、オフチップメモリの値の変更(図 1(c))や、オフチップメモリのアドレスの変更(図 1 (d))がされた場合、オフチップメモリとレジスタの間で整合性がとれなくなる危険性がある。本文ではプロファイリングとデータの依存関係調査を用い、この整合性を保ちつつ、レジスタ化を行うことでオフチップへのメモリアクセスを削減する。以降、直接メモリの値を変更することと、オフチップメモリのアドレスを変更することの両方を「メモリライト」と呼ぶ。

†近畿大学　理工学部電気電子工学科
‡近畿大学　総合理工学研究科　エレクトロニクス系工学専攻

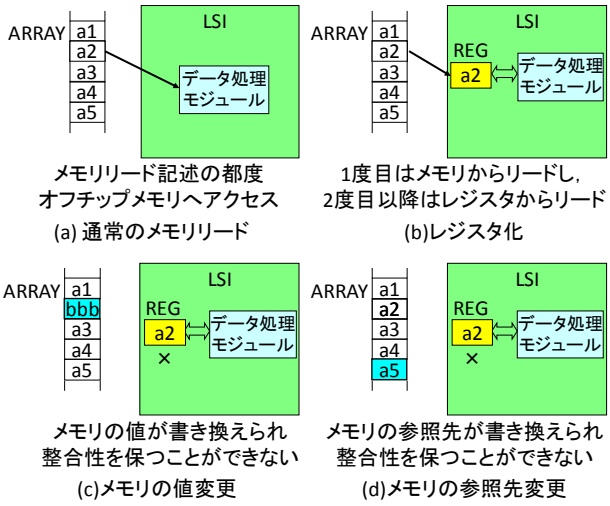

図 1　レジスタ化とメモリライト

2.2.　整合性対策

整合性対策とは、アドレス変数が変化した場合にオフチップメモリとレジスタ間でデータの整合性を保つこととする。この不整合は図 1(c)でも発生するが、異なるアドレス変数(以降：「更新アドレス変数」)によるメモリライトがされた場合は、異なる対処が必要である。すなわち、更新アドレス変数によるメモリライトの場合、レジスタに格納しているオフチップメモリのアドレス(以降：「格納アドレス変数」)の値が更新されたか、別のアドレスの値が更新されたかわからない。そのため以下の手順を踏み、レジスタの更新を行う。

プログラム実行時に格納アドレス変数と更新アドレス変数の一致をチェックし、一致していたらオフチップメモリの値をレジスタに格納する処理を回路に追加記述する。この対策を「整合性対策」と呼ぶ。

2.3.　再利用データのオンチップ配列化

2.1 節では、複数の単変数のレジスタ化を考えたが、複数個の再利用データをオンチップに格納するにあたって、ループ構造などオフチップへのメモリアクセスがパターン化(アクセスの順番が固定化)される場合が画像処理などで多く存在する。

図 2 の場合、w_l という変数が一定期間変化しない間、オフチップのある範囲(右図では array によって決まる探索範囲)に対してループ構造による複数回の探索（図は二分探索の例）が行われる。このような場合の再利用される複数データをオンチップ配列化することにより、オフチップへのメモリアクセスを削減することを考える。

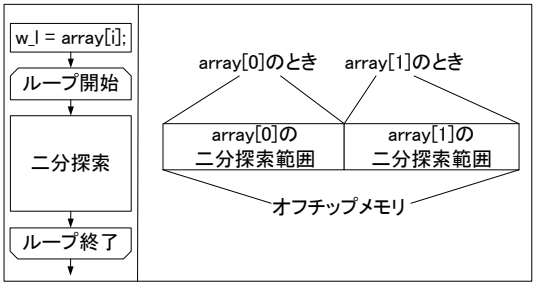

図 2 オフチップメモリの探索の制限例

2.4. オンチップ配列への格納と参照

オフチップからオンチップ配列にデータを格納する(以降：「配列格納」)ために以下の方法を考える。2.3 節で述べたようにループ構造におけるオフチップへのメモリアクセスがパターン化されることに注目し、配列格納は以下のパターンに沿って行う。

①オフチップからオンチップメモリへ格納するループ構造による複数データがある時、処理手順(アドレス変更がある処理手順)を基にオンチップ配列へデータを格納する。
②オンチップ配列へのアクセスアドレスは①で求めた経路順に決定する。

2.5. オンチップ配列に対するメモリライト

オフチップメモリにデータ更新があった場合、整合性を保つためにオンチップ配列中のデータも更新しなくてはならない。しかし、オンチップ配列化では 2.1 節のレジスタ化と異なり複数のデータをメモリに保持しているため、オンチップ配列を更新するアドレスを探索する問題がある。オンチップ配列のデータ更新の概念図を図 3 に示し、以下に方法を示す。

1.オンチップ配列のアドレスとオフチップメモリのアドレスの対応表を配列格納時にオンチップに保持する。
2.対応表をソートする。
3.データ更新時には対応表を探索し、オフチップアドレスからオンチップ配列アドレスを求め、データ更新する。
4.同じオフチップデータが複数ある場合があるので前後の対応表も探索し、同じデータなら更新する。

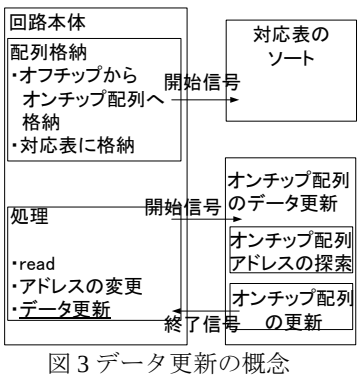

図 3 データ更新の概念

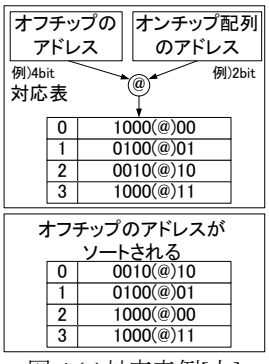

図 4 (a)対応表例[上]
(b)ソート例[下]

図 3 の概念図のように、回路本体と上記の 2、3 と 4 の処理を並列に実行する。図 4 にオンチップ配列とオフチップのアドレス対応表の格納とソートの例を示す。対応表は図 4(a)に示している通り、オフチップのアドレスとオンチップ配列のアドレスをビット連結し、格納する。これにより上記の方法 2 でソートを行うことで図 4(b)のようにオフチップのアドレスがソートされる。

3. レジスタ化の回路自動生成手法

レジスタ化手法を回路へ自動的に適用させるソフトウェアを作成した。そのアルゴリズムを図 5 に示す。

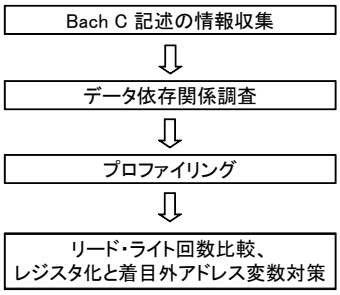

図 5 自動レジスタ化のアルゴリズム

3.1. Bach C 記述の情報収集

Bach C 記述からの情報収集ではデータ依存関係調査の処理を行う際に必要となる情報を Bach C 記述から集めている。具体的には、オフチップメモリに割り当てられた配列の情報、制御文の情報、関数呼び出し、メモリのアドレスが変更される記述行の情報、オフチップメモリの値が変更される記述行の情報等を収集している。これら収集した情報から図 6 に示すグラフを作成する。

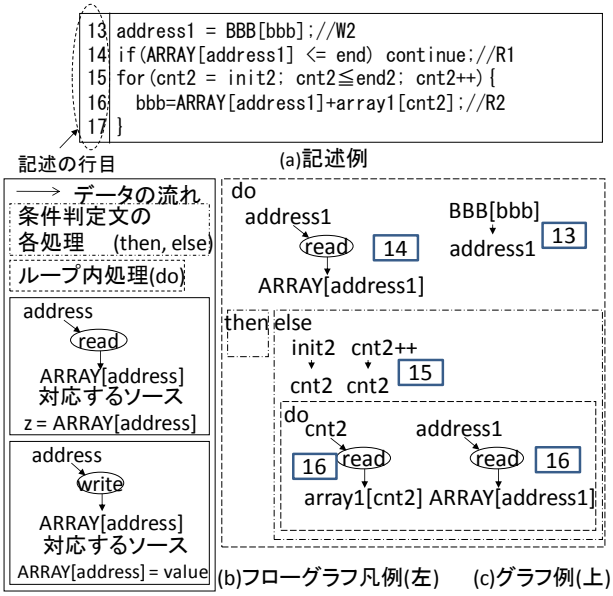

図 6 Bach C 記述からの情報

図 6(a)の記述例へのアルゴリズムの適用について述べる。図 6(b)は(c)のグラフの凡例である。また図 6(c)でリード、ライトノードで表記されている配列はすべてオフチップメモリであり、四角で囲まれた数字が記述行を示している。図 6(c)は、各々の記述がどの制御文内に存在し、リー

ドまたはライトされているかを示している。例えば記述16行目の ARRAY[address1]は for 文->else 文->for 文の制御文内にあり、リードがされていることを情報として保持している。

3.2. データ依存関係調査

データ依存関係調査では 3.1 節で収集した Bach C 記述からの情報を基に、メモリからレジスタへデータを格納する記述箇所を調べる。これは、レジスタとメモリのデータの整合性を保つための処理である。

レジスタ化を施す決定は各メモリリードに対して行うため、メモリリードごとにデータ依存関係を調査する。ここでは、メモリリードに対してレジスタ化の効果を最大限に生かすよう最低限のメモリアクセスを目指している。そのため、データの依存関係の大きいメモリリードの記述行より前方のメモリライトから調査を行い、次いで後方のメモリライトの調査を行う。これらの調査を行った結果は、図 6 にデータ線(太い矢印線)を加えた図 7 のフローグラフとなる。

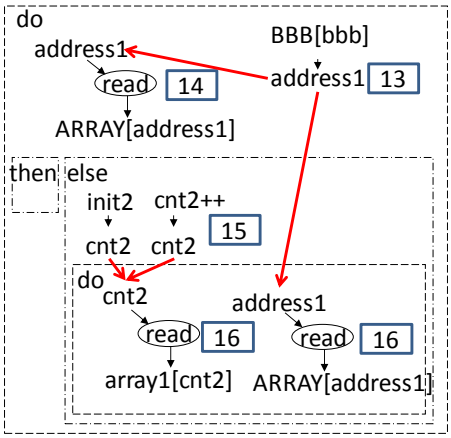

図 7 データ依存関係を含むフロー

図 7 の太い矢印線がデータ依存関係を示している。14 行目の ARRAY[address1]は 13 行目の address1 のメモリライトと依存関係があることを示している。14 行目の ARRAY[address1]がレジスタ化を施すことに決定した場合、13 行目でメモリからレジスタにデータを格納する記述を追加する。

3.3. プロファイリング

プロファイリングにより記述のメモリリード、ライトが何回実行されているか測定する。Bach システムは記述の各行の実行回数をカウントする「bachprof」というプロファイリングツールを備えている。この bachprof を実行すると図 8 のように実行回数が出力される。

各行の実行回数

```
100  13 address1 = BBB[bbb];//W2
100  14 if (ARRAY[address1] <= end) continue;//R1
 70  15 for (cnt2 = init2; cnt2≦end2; cnt2++){
700  16   bbb = ARRAY[address1] + array1[cnt2];//R2
     17 }
```

図 8 プロファイル結果

3.4. リード・ライト回数比較とレジスタ化

リード・ライト回数比較では 3.2 節のデータ依存関係調査、3.3 節のプロファイリングの結果を基に各メモリリードについてレジスタ化を施すかを決定する。

(1) あるメモリリードに着目し、そのメモリリードの実行回数と依存関係があるメモリライトの実行回数をプロファイリング結果から抽出する。複数メモリライトがある場合や同じ依存関係のメモリリードは実行回数の和をとる。

(2) 以下の式が成立すれば、レジスタ化を決定する。
リード回数＞依存関係にあるメモリライト回数

(3) レジスタ化が決定した場合、依存関係にあるメモリライトはレジスタ更新を行う。この後、着目したメモリリードと同じ配列、アドレス変数のメモリリードのリード・ライト回数を比較する際は、今回依存関係にあったメモリライトの実行回数を 0 回とする。

(4) これらの操作を各メモリリードに対して行う。リード・ライト回数比較で、ある着目したメモリリードの実行回数が、メモリライトの実行回数を上回るものがなくなるまで(1)～(4)の処理を繰り返す。

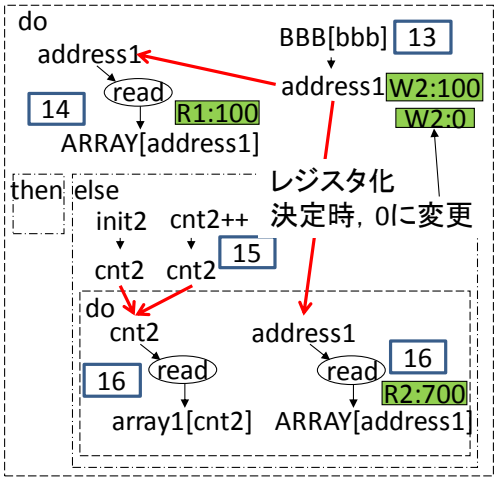

図 9 リード・ライト回数比較時のフロー変化

図 9 の R1,R2,W2 の右に示している数字は 3.3 節で得た各行の実行回数である。ここでリード・ライト回数比較について例を用いて示す。

(1) 14 行目の ARRAY[address1]のメモリリードに着目し、依存関係である 13 行目のメモリライトの実行回数を取り出す(100 回)。また、同じ依存関係のメモリリードが 16 行目にあるので、和をとる。

(2) 100 回(14 行目) + 700 回(16 行目) ＞ 100 回(13 行目)
比較式が成り立つのでレジスタ化が決定。

(3) 次から ARRAY[address1]に対する 13 行目のメモリライトの実行回数を 100 回から 0 回に変更する。

以上の処理を各メモリリードに対して行う。

図 6(a)の記述に対し、レジスタ化を施した結果は図 10 となる。

```
13  address1 = BBB[bbb];//W2
    ARRAY_reg = ARRAY[address1];//レジスタ更新
14  if (ARRAY_reg <= end) continue;//R1
15  for (cnt2 = init2; cnt2≦end2; cnt2++){
16    bbb = ARRAY_reg + array1[cnt2];//R2
17  }
```

図 10 レジスタ化後の記述

図 10 は図 7(a)でのリード・ライト回数比較の結果より、14,16 行目の ARRAY[address1]に対してレジスタ化を施すことが決定したので、そこをレジスタ ARRAY_reg に書き換えている。また、それぞれのデータ依存関係は 13 行目であるので 13 行目にレジスタの更新を追加し、整合性を保っている。

4. 実験結果とその評価

レジスタ化は音声認識技術におけるビタビ探索に、オンチップ配列化は同技術のバイグラム探索に対し適用した結果とその評価を以下に示す。

4.1. レジスタ化の回路自動生成の適用

レジスタ化をビタビ探索に適用した。検証の結果、手動で行ったレジスタ化とソフトウェアを用いたレジスタ化の記述が一致した。また、レジスタ化を施す前後でのビタビ探索の出力結果が一致していたことから、レジスタ化が問題なく変換されていることを確認した。

ビタビ探索に対して、着目したメモリリード記述箇所が 53 箇所あり、その中からリード・ライト比較により実際にレジスタ化を施した記述箇所が 27 箇所であった。また、レジスタ化のために新たに用意したレジスタの数は 7 つで、整合性対策のために用意したフラグは 2 つであった。

4.2. オンチップ配列化の適用

オンチップ配列化をバイグラム探索に対し、2.3 節、2.4 節で述べた手法を適用した。バイグラム探索は図 2 に示したように w_l が一定の間に複数回の二分探索がある。バイグラム探索に対し、オンチップ配列を 127 要素用意した。この例ではメモリライトが存在しない。

4.3. レジスタ化の測定結果

レジスタ化を施す前後のビタビ探索の 100 フレームの平均処理時間と回路規模の増減を表 1 に示す。動作周波数は 100MHz、オフチップメモリへのアクセス時間を 50ns とした。

表 1　レジスタ化によるビタビ探索の処理向上率

	平均処理時間[ns]	回路規模[ゲート]
レジスタ化前	56,951,607	48,896
レジスタ化後	49,341,193	48,928
処理向上率	13.36	

表 1 の結果より、レジスタ化を施す前と後では処理速度は約 13.36%向上し、回路規模は 32 ゲート増加にとどまった。処理速度の向上はオフチップメモリへのアクセス回数が 10,393,929 回から 1,959,242 回(約 18.85%)に減らせたためであり、回路規模の増加は新たにレジスタを 9 つ用意したことが原因と考えられる。

4.4. オンチップ配列化の測定結果

オンチップ配列化をバイグラム探索に適用させた場合の見積もり計算を行った。この時の、100 フレーム分の処理時間と処理向上率を表 2 に示す。動作周波数は 100MHz、オフチップメモリへのアクセス時間を 50ns、オンチップ配列へのアクセス時間を 5ns とした。

表 2　オンチップ化の処理向上率

要素数	処理時間[ns]		向上率[%]
	オンチップ化前	オンチップ化後	
127	2,247,355,150	1,478,066,470	34.23

表 2 の結果より、オンチップ配列化を施す前と後では処理速度は約 34.23%向上した。処理速度の向上はオフチップメモリへのアクセス回数が 26,619,711 回から 5,368,915 回(20.18%)に減らせたためと考えられる。

5. まとめと今後の課題

メモリアクセスの高速化を考える上で、レジスタ化では再利用する値をレジスタに格納したが、キャッシュメモリに格納する方法も考えられる。しかし、キャッシュメモリはチップ内にデータを格納するため、データ収得に一度アドレスを送信しなければならず、参照データの収得に時間がかかり、かつオフチップメモリとデータ一致を図るため複雑な機構が必要となる。よって今回はレジスタを利用した。今後はメモリライトを含むオンチップ配列化についてプログラムを実装し、実験評価し、レジスタ化を含めた統合メモリアクセスアーキテクチャ生成技術を開発していく。

謝辞

Bach を用いたハードウェア設計を実現するに当たり、多大なる御指導を頂いたシャープ株式会社 IC 事業本部要素技術開発センター山田晃久様をはじめ、BACH 開発グループの皆様に心から御礼申し上げます。

本研究は、東京大学大規模集積システム設計教育研究センターを通し、シノプシス株式会社の協力で行われたものである。

参考文献

[1]“Bach システムマニュアル”；シャープ株式会社提供、2003 年

[2]富田眞治; “コンピュータアーキテクチャ”、丸善、2000 年

[3]A. Eguchi, *et al.*:”A Hardware Design for the First Pass of A Large Vocabulary Continuous Speech Recognition System,” The proceeding of 15th SASIMI, pp. 230 - 235, 2009.

[4]I. Issenin and N. Dutt: “A Data Reuse Analysis Technique for Efficient Scratch-Pad Memory Management,” ACM Trans. on DA of Electronic System, Vol. 12,No. 2-15, April 2007.